

There is no fast lunch: an examination of the running speed of evolutionary algorithms in several languages

J.J. Merelo^{*1}, P. García-Sánchez^{†1}, M. García-Valdez^{‡2}, and I. Blancas^{§1}

¹University of Granada, Spain

²Instituto Tecnológico de Tijuana, Mexico

Abstract

It is quite usual when an evolutionary algorithm tool or library uses a language other than C, C++, Java or Matlab that a reviewer or the audience questions its usefulness based on the speed of those other languages, purportedly slower than the aforementioned ones. Despite speed being not everything needed to design a useful

evolutionary algorithm application, in this paper we will measure the speed for several very basic evolutionary algorithm operations in several languages which use different virtual machines and approaches, and prove that, in fact, there is no big difference in speed between interpreted and compiled languages, and that in some cases, interpreted languages such as JavaScript or Python can be faster than compiled languages such as Scala, making them worthy of use for evolutionary algorithm experimentation.

1 Introduction

It is a well extended myth in scientific programming to claim that compiled languages such as C++ or Java are always, in every circumstance, faster than interpreted languages such as Perl, JavaScript or Python.

However, while it is quite clear that efficiency matters, as said in [1], in general and restricting the concept of *speed* to *speed of the compiled/interpreted application* it might be the case that some languages are faster to others, as evidenced by benchmarks such as [2, 3]. Taken in general or even restricting it to some particular set of problems such as floating point computation, some compiled languages tend to be faster than interpreted languages.

But, in the same spirit of the *There is no free lunch* theorem [4] we can affirm there is a *no fast lunch* theorem for the implementation of evolutionary

*jmerelo@ugr.es. Reachable also at the issues section of the repository for this paper.

†fergunet@gmail.com

‡mariosky@gmail.com

§iblancasa@gmail.com

optimization, in the sense that, while there are particular languages that might be the fastest for particular problem sizes and specially fitness functions, in general the fastest language will have those two dependencies, and, specially, for non-trivial problem sizes and limiting ourselves to the realm of evolutionary algorithm operators, scripting languages such as JavaScript might be as fast or even faster than compiled languages such as Java.

Coming up next, we will write a brief state of the art of the analysis of implementations of evolutionary algorithms. Next we will present the test we have used for this paper and its rationale, and finally we will present the results of examining four different languages running the most widely used evolutionary algorithm operator: mutation. Finally, we will draw the conclusions and present future lines of work.

2 State of the art

In fact, the examination of the running time of an evolutionary algorithm has received some attention from early on. Implementation matters [5, 6], which implies that paying attention to the particular way an algorithm is implemented might result in speed improvements that outclass that achieved by using the *a priori* fastest language available. In fact, careful coding led us to prove [7] that Perl, an interpreted and not optimized for speed language, could obtain times that were on the same order the magnitude as Java. However, that paper also proved that, for the particular type of problems used in scientific computing in general, the running speed is not as important as coding speed or even learning speed, since most scientific programs are, in fact, run a few times while a lot of time is spent on coding them. That is why expressive languages such as Perl, JavaScript or Python are, in many cases, superior to these fast-to-run languages.

However the benchmarks done in those papers were restricted to particular problem sizes. Since program speed is the result of many factors, including memory management and implementation of loop control structures, in this paper we will examine how fast several languages are for different problem sizes. This will be done next.

3 Experimental setup

First, a particular problem was chosen for testing different languages and also data representations: performing bit-flip mutation on a binary string. In fact, this is not usually the part of the program an evolutionary algorithm spends the most time in [6]. In general, that is the fitness function, and then reproduction-related functions: chromosome ranking, for instance. However, mutation is the operation that is performed the most times on every evolutionary algorithm and is quintessential to the algorithm itself, so it allows the comparison of the different languages in the proper context.

Essentially, mutation is performed by

1. Generating a random integer from 0 to the length of the chromosome.
2. Choosing the bit in that position and flipping it
3. Building a chromosome with the value of that bit changed.

Chromosomes can be represented in at least two different ways: an array or vector of boolean values, or any other scalar value that can be assimilated to it, or as a bitstring using generally “1” for true values or “0” for false values. Different data structures will have an impact on the result, since the operations that are applied to them are, in many cases, completely different and thus the underlying implementation is more or less efficient.

Then, four languages have been chosen for performing the benchmark. The primary reason for choosing these languages was the availability of open source implementations for the author, but also they represent different philosophies in language design.

Language	Version	URL
Scala	2.11.7	http://git.io/bfscala
Lua	5.2.3	http://git.io/bflua
Perl	v5.20.0	http://git.io/bfperl
JavaScript	node.js 5.0.0	http://git.io/bfnode
http://git.io/bfpython Python	2.7.3	http://git.io/bfpython

Table 1: Languages used and file written to carry out the benchmark. No special flags were used for the interpreter or compiler.

Compiled languages are represented by Scala, a strongly-typed functional language that compiles to a Java Virtual Machine bytecode. Scala is in many cases faster than Java [3] due to its more efficient implementation of type handling. Two different representations were used in Scala: `String` and `Vector[Boolean]`. They both have the same underlying type, `IndexedSeq` and in fact the overloading of operators allows us to use the same syntax independently of the type. The benchmark, `bitflip.scala`, is available under a GPL license, at the URL shown in Table 1.

Interpreted languages are represented by Lua, Perl and Javascript. Lua is a popular embedded language that is designed for easy implementation; Perl has been used extensively for evolutionary algorithms [7, 8, 9] with satisfactory results, and node.js, an implementation of JavaScript, which uses a the V8 JIT compiler to create bytecode when it reads the script, and has been used lately by our research group as part of our NodeIO library [10] and volunteer computing framework NodIO [11]. In fact, this paper is in part a rebuttal to concerns made by reviewers of the lack of speed and thereof adequacy of JavaScript for evolutionary algorithm experimentation. Versions and files are shown in the Table 1. Only Perl used two data structures as in Scala: a string, which is a scalar structure in Perl, and an array of booleans.

In all cases except in Scala, implementation took less than one hour and was inspired by the initial implementation made in Perl. Adequate data and control structures were used for running the application, which applies mutation to a single generated chromosome a hundred thousand times. The length of the mutated string starts at 16 and is doubled until 2^{15} is reached, that is, 32768. This upper length was chosen to have an ample range, but also so small as to be able to run the benchmarks within one hour. Results are shown next.

4 Results and analysis

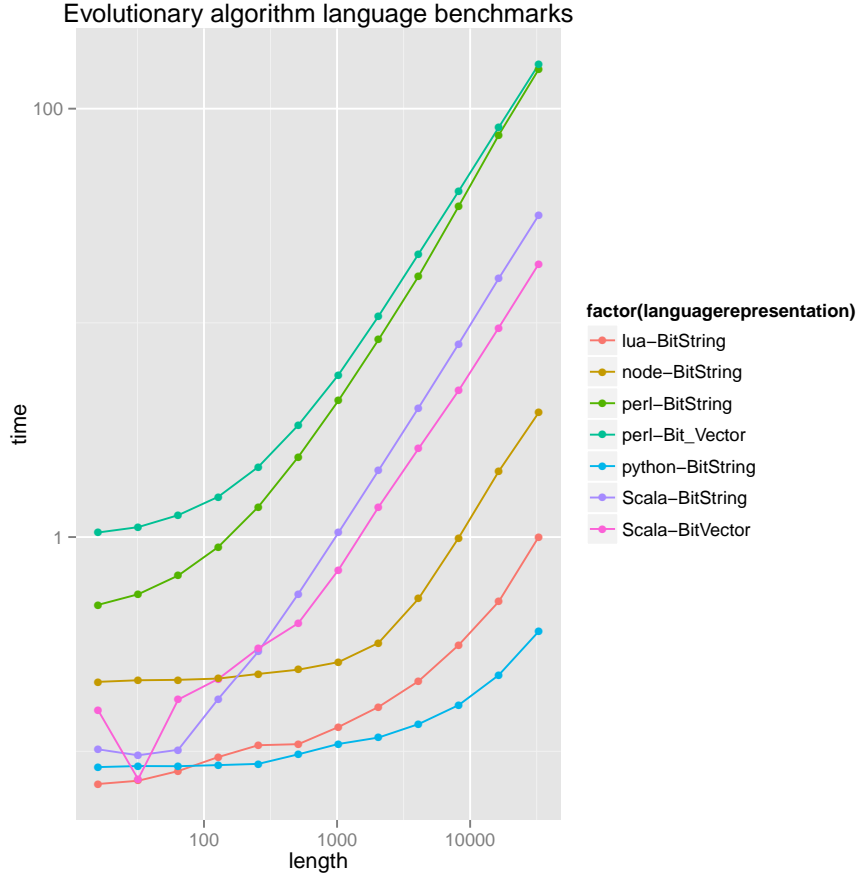


Figure 1: Plot of time needed to perform 100K mutations in strings with lengths increasing by a factor of two from 16 to 2^{15} . Please note that x and y both have a logarithmic scale.

All measurements and processing scripts are included in this paper repository, although in fact the programs were written to directly produce a CSV (comma separated value) representation of measurements, which was then plotted using R and `ggplot` as shown in Figure 1. The first unexpected behavior shown here is the remarkable speed of the Lua language, which is, in fact, faster than any other small sizes, although slower than Python at bigger sizes. After these two, next position goes to node.js, which uses a very efficient implementation of a Just-in-Time interpreter for the JavaScript language. Then Scala, whose bitvector representation is better than the bitstring and finally Perl, with a bitstring representation being slightly better than bit vectors, although the difference dilutes with time. In fact, Scala is a bit better than node for the smaller sizes, less than 128 bits and any representation, but that advantage disappears for greater sizes.

The behavior of the languages is not linear either. Node.js and Python both have an interesting feature: its speed is roughly independent of the string size up to size 1024. The same happens also for several size segments for Lua, showing some plateaus that eventually break for the bigger sizes. It probably means that the creation and accessing of strings is done in constant time, or is roughly constant, giving it a performance advantage over other languages. Even so, it never manages to beat the fastest language in this context, which is either Lua or Python, depending on the size.

The trend from size 1024 on is for the differences to keep in more or less the same style for bigger sizes, so we do not think it would be interesting to extend it to 2^{16} and upwards. In any case, these measures allow us to measure the performance of the most widely used genetic operator in four different and popular languages, since all four of them (except for Lua) show up in most rankings of the most popular languages, such as the Tiobe ranking [12].

5 Conclusions

In this paper we set out to measure the speed of different languages when running the classical evolutionary algorithm operation: mutating a chromosome represented as a binary string or vector. The results can be a factor on the choice of a language for implementing solution to problems using evolutionary algorithms, at least if raw running speed is the main consideration. And, if that is our main concern, the fastest language for mutating strings has been found to be Lua and Python, followed by node.js and Perl. Most interpreted languages are faster for the wider range of chromosome sizes than Scala, which is a compiled language that uses the Java Virtual machine.

Despite its speed Lua is not exactly a popular language, although it definitely has found its niche in embedded systems and applications such as in-game scripting, game development or servers so our choice for EA programming languages would be Python or JavaScript, which are fast enough, popular, allow for fast development and have a great and thriving community. JavaScript does have an advantage over Python: Besides being interpreted languages and using dynamic typing, it can express complex operations in a terse syntax and bestows implementations both in browsers and on the server. We can conclude from these facts and the measurements made in this paper that JavaScript is perfectly adequate for any scientific computing task, including evolutionary algorithms.

That does not mean that Perl or Scala are not adequate for scientific computing. However, they might not be if experiments take a long time and time is of the essence; in that case implementing the critical parts of the program using C, Go, Lua or Python might be the right way to go. But in general, it cannot be said that interpreted languages are not an adequate platform for implementing evolutionary algorithms, as proved in this paper.

Future lines of work might include a more extensive measurement of other operators such as crossover, tournament selection and other selection algorithms. However, they are essentially CPU integer operations and their behavior might be, in principle, very similar to the one shown here. This remains to be proved, however, but it is left as future line of work.

6 Acknowledgements

This paper is part of the open science effort at the university of Granada. It has been written using `knitr`, and its source as well as the data used to create it can be downloaded from the GitHub repository. It has been supported in part by GeNeura Team.

This work has been supported in part SPIP2014-01437 (Dirección General de Tráfico), PRY142/14 (Fundación Pública Andaluza Centro de Estudios Andaluces en la IX Convocatoria de Proyectos de Investigación), TIN2014-56494-C4-3-P (Spanish Ministry of Economy and Competitvity), and PYR-2014-17 GENIL project (CEI-BIOTIC Granada).

References

- [1] E. Anderson, J. Tucek, Efficiency matters!, ACM SIGOPS Operating Systems Review 44 (1) (2010) 40–45.
- [2] L. Prechelt, An empirical comparison of seven programming languages, Computer 33 (10) (2000) 23–29.
- [3] B. Fulgham, I. Gouy, The computer language benchmarks game (2010).
URL <http://shootout.alioth.debian.org>
- [4] D. H. Wolpert, W. G. Macready, No free lunch theorems for optimization, IEEE Transactions on Evolutionary Computation 1 (1) (1997) 67–82.
URL citeseer.nj.nec.com/wolpert96no.html
- [5] J.-J. Merelo-Guervós, G. Romero, M. García-Arenas, P. A. Castillo, A.-M. Mora, J.-L. Jiménez-Laredo, Implementation matters: Programming best practices for evolutionary algorithms, in: J. Cabestany, I. Rojas, G. J. Caparrós (Eds.), IWANN (2), Vol. 6692 of Lecture Notes in Computer Science, Springer, 2011, pp. 333–340.
- [6] S. Nesmachnow, F. Luna, E. Alba, Time analysis of standard evolutionary algorithms as software programs, in: Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on, IEEE, 2011, pp. 271–276.
- [7] J.-J. Merelo-Guervós, P.-A. Castillo, E. Alba, `Algorithm::Evolutionary`, a flexible Perl module for evolutionary computation, Soft Computing 14 (10) (2010) 1091–1109, accesible at <http://sl.ugr.es/000K>.
doi:10.1007/s00500-009-0504-3.
URL <http://www.springerlink.com/content/8h025g83j0q68270/fulltext.pdf>
- [8] J. J. Merelo, P. A. Castillo, A. Mora, A. Fernández-Ares, A. I. Esparcia-Alcázar, C. Cotta, N. Rico, Studying and tackling noisy fitness in evolutionary design of game characters, in: A. Rosa, J. J. Merelo, J. Filipe (Eds.), ECTA 2014 - Proceedings of the International Conference on Evolutionary Computation Theory and Applications, 2014, pp. 76–85.

- [9] J. J. M. Guervós, A. M. Mora, P. A. Castillo, C. Cotta, M. G. Valdez, A search for scalable evolutionary solutions to the game of mastermind, in: IEEE Congress on Evolutionary Computation, IEEE, 2013, pp. 2298–2305.
- [10] J.-J. Merelo-Guervós, P.-A. Castillo-Valdivieso, A. Mora-García, A. Esparcia-Alcázar, V.-M. Rivas-Santos, NodEO, a multi-paradigm distributed evolutionary algorithm platform in JavaScript, in: D. V. Arnold, E. Alba (Eds.), Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014, Companion Material Proceedings, ACM, 2014, pp. 1155–1162. doi:10.1145/2598394.2605688.
URL <http://doi.acm.org/10.1145/2598394.2605688>
- [11] J. J. M. Guervós, P. García-Sánchez, Modeling browser-based distributed evolutionary computation systems, CoRR abs/1503.06424.
URL <http://arxiv.org/abs/1503.06424>
- [12] TIOBE, Tiobe index for october 2015, Tech. rep., TIOBE (October 2015).
URL <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>